Project 3

Name: Haopeng Chen

ID: 120090645

How to Start

```
cmake -B build
cmake --build build -j 16
sh src/sbach.sh
```

Design & Analysis

Task 1

We simply partition the data into several blocks and send each data block to one MPI process, which performs sequential quick-sort inside to make that data block in order. After all the processes finish sorting their data, all sorted sub-arrays are sent back the master process and construct the result by doing k-way merging using a binary heap.

Suppose we have k processes and an array with length of n. Every time we only using the smallest remaining element in each array, that is, k elements for k processes and put them into the binary heap. After poping out the smallest one in these k items, we insert another one from the array that poped item originated from. We keep doing this until all sub-arrays are traversed. The binary heap gives out $O(k \log k)$ for one insertion/pop out. As we will traverse n items, it should be $O(k \log k \cdot n)$. In our experiments, $k \leq 32$, it turns out to be $O(n)$.

Therefore, the dominated time is still consumed in quicksort, giving $O(\frac{n}{k} \log \frac{n}{k})$ in average and $O(\frac{n^2}{k^2})$ in worst case. It is still $O(n \log n)$ and $O(n^2)$, respectively. However, for actual cases and our experiments, the merging time also matters for large k.

# Process(es)	1	2	4	8	16	32
Execution Time / ms	14112	13406	9725	8856	7928	9307
K-way Merge Time / ms	0	2457	3683	4906	6033	7509
Speedup	1	1.05	1.45	1.59	1.78	1.52
Efficiency	1	0.53	0.73	0.80	0.89	0.76

Task 2

We simply partition the data into several blocks and send each data block to one MPI process, which performs sequential bucket-sort inside to make data in several buckets. Then we apply sequential insertion sort to each bucket one by one in each process. After all the processes finish sorting their data, all sorted sub-arrays are sent back the master process and construct the result by doing k-way merging using a binary heap.

Suppose we have k processes and an array with length of n. Every time we only using the smallest remaining element in each array, that is, k elements for k processes and put them into the binary heap. After poping out the smallest one in these k items, we insert another one from the array that poped item originated from. We keep doing this until all sub-arrays are traversed. The binary heap gives out $O(k \log k)$ for one insertion/pop out. As we will traverse n items, it should be $O(k \log k \cdot n)$. In our experiments, $k \leq 32$, it turns out to be $O(n)$.

Therefore, the dominated time is still consumed in bucket sort. In our design of buckets, elements are distributed into each bucket generally uniformly. We could only care average case in this experiment. Consider we have b buckets in total. Therefore, in each process, we have $\frac{n}{k}$ elements and $\frac{b}{k}$ buckets. Scattering each element into buckets costs $O(\frac{n}{k})$. Insertion sort in each bucket costs $O(\frac{n^2}{b^2})$ and we do each bucket sequentially. As mentioned above, $k \leq 32$, the total time should be $O(n + \frac{n^2}{b^2} \cdot b) = O(\frac{n^2}{b})$. Here gives out the expected experiment results, buckets number b significantly impact the execution time, especially b is comparable with n.

# Process(es)	**1**	**2**	**4**	**8**	**16**	**32**
Execution Time / ms	14777	24458	16319	11130	8877	8574
Speedup	1	0.60	0.90	1.32	1.66	1.72
Efficiency	1	0.30	0.23	0.17	0.10	0.05

Task 3

In MPI, each process has its own memory space and cannot access the data of other processes. Therefore, when the comparison encounters the boundary of a subarray, communication is required among the workers to determine whether an exchange is needed. Another important aspect is that, at the end of each iteration, every process must inform the master process whether its local sub-vector is sorted or not. If the master process determines that all the sub-vectors are sorted, it then informs all the processes that the vector has been sorted, and it's time to send the sub-vectors back to form a complete one.

Consider the worse case on an array with length of n. The smallest element is placed on the end. Each time we do n pairs comparison through the whole array as one phase, and we need n times for such an *sorting phase* to put this smallest element at the beginning. Obviously, the sequential runtime of odd-even sort is $O(n^2)$. And suppose we have k processes, we are doing is similar but on $\frac{n}{k}$ elements. It gives out $O(\frac{n^2}{k^2})$, but as we consider $k \leq 32$, it is still $O(n^2)$ generally.

# Process(es)	**1**	**2**	**4**	**8**	**16**	**32**
Execution Time / ms	42809	15657	3918	982	246	65
Speedup	1	2.73	10.93	43.59	174.02	658.60
Efficiency	1	1.37	2.73	5.45	10.88	20.58